Évaluation et optimisation de requêtes

Serge Abiteboul
à partir de tranparents de Philippe Rigaux, Dauphine

INRIA Saclay

April 3, 2008

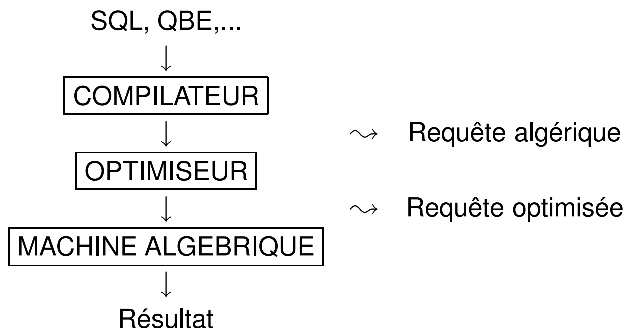
But de ce cours

Soit une requête. Le but est de l'évaluer efficacement.

- Comment on passe de SQL (déclaratif) à un arbre d'opérations (algébrique).
- Comment **l'optimiseur** obtient plusieurs plans d'exécution possibles.
- Comment il effectue un choix parmi ces plans.
- Comment la requête est ensuite évaluée.

Ce sont des techniques de base, implantées dans tout SGBD relationnel, et dans ORACLE en particulier.

Séquencement



Opérations

1. **compilateur** : traduction de la requête dans l'algèbre
2. **optimiseur** : trouver un plan d'exécution tel que le coût d'exécution soit minimal

Étapes du traitement d'une requête

Toute requête SQL est traitée en trois étapes :

1. **Analyse et traduction** de la requête. On vérifie qu'elle est correcte, et on la réécrit sous forme de requête algébrique avec des opérations.
2. **Optimisation** :
   - On le réécrit en requêtes équivalentes
   - On annote les opérateurs avec des choix d'algorithmes particuliers
   - On obtient des **plans d'exécution** dont on évalue les coûts
   - On choisit le "meilleur".
3. **Exécution de la requête** : le plan d'exécution est compilé et exécuté.

La base mathématique: Équivalences algébriques

Commutativité et associativité de la jointure

$$E_1 \bowtie E_2 = E_2 \bowtie E_1,$$
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3).$$

Cascade de projections

$$\pi_{A_1,\ldots,A_n}(\pi_{B_1,\ldots,B_m}(E)) = \pi_{A_1,\ldots,A_n}(E)$$

Cascade de sélections

$$\sigma_{F_1}(\sigma_{F_2}(E)) = \sigma_{F_1 \wedge F_2}(E)$$

Commutation sélection et projection

Si F ne porte que sur $A_1,\ldots,A_n$,

$$\pi_{A_1,\ldots,A_n}(\sigma_F(E)) = \sigma_F(\pi_{A_1,\ldots,A_n}(E))$$

Si F porte aussi sur $B_1,\ldots,B_m$,

$$\pi_{A_1,\ldots,A_n}(\sigma_F(E)) = \pi_{A_1,\ldots,A_n}(\sigma_F(\pi_{A_1,\ldots,A_n,B_1,\ldots,B_m}(E)))$$

Équivalences algébriques (2)

Commutation sélection et $\times \cup - \bowtie$

$$\sigma_F(E_1 OP E_2) = \sigma_F(E_1) OP \sigma_F(E_2)$$

Commutation projection et $\times \cup$

$$\pi_{A_1,\ldots,A_n}(E_1 OP E_2) = \pi_{A_1,\ldots,A_n}(E_1) OP \pi_{A_1,\ldots,A_n}(E_2)$$

Équivalence logique

Exemple
SELECT *
FROM VIN V
WHERE ((V.DEGRE = 12) OR (V.CRU = 'Morgon') OR (V.CRU = 'Chenas'))
AND NOT ((V.CRU = 'Morgon') OR (V.CRU = 'Chenas'));

Propriété logique: $((P \vee Q \vee R) \wedge \neg(Q \vee R)) \Leftrightarrow (P \wedge \neg(Q \vee R))$

Expression simplifiée de la requête
SELECT *
FROM VIN V
WHERE ((V.DEGRE = 12)
AND NOT ((V.CRU = 'Morgon') OR (V.CRU = 'Chenas'));

Equivalence : contraintes d'intégrité

EXEMPLE1

$$Q_1'' = \pi_{v\#,nom}(vin) \bowtie \pi_{v\#,cépage}(vin)$$

Si $v\# \rightarrow nom$

$Q_1'' \approx vin$ et la jointure est inutile

EXEMPLE2 : REQUETE SQL

SELECT *

FROM VIN

WHERE (CRU = 'Jurançon') AND (DEGRE $\leq$ 10);

CONTRAINTE : CRU = 'Jurançon' $\Rightarrow$ DEGRE $\geq$ 12

SELECT * FROM VIN

WHERE (CRU = 'Jurançon') AND (DEGRE $\leq$ 10) AND (DEGRE $\geq$ 12)

Réponse vide

Quelle optimisation ?

Optimiser quelles ressources : processeurs, **accès disques**, communication (BD distribuées)

Optimiser quoi : le **temps de réponse** à une requête; le nombre de requêtes traitées par unité de temps - **débit**.

On demande en général que le temps d'optimisation soit négligeable par rapport à l'exécution de la requête.

- Coût de l'optimisation: faible - peu d'accès disques
- Gain: accès disque à l'exécution

Quelle information on peut utiliser ?

Le traitement s'appuie sur les éléments suivants :

1. **Le schéma logique de la base**, description des tables, des contraintes d'intégrité
2. **Le schéma physique de la base**, indexes et chemins d'accès, tailles des blocs
3. **Des statistiques** : taille des tables, des index, distribution des valeurs
   R_1 10000 nuplets (site A) et R_2 200 nuplets (site B)
   pour réaliser $R_1 \bowtie R_2$ - il vaut mieux transférer R_2
4. **Des statistiques** : taux de mises-à-jour, workload
5. **Les particularités du système** : parallélisme, processeurs spécialisés (e.g. filtre)
6. **Des algorithmes** : il peuvent différer selon les systèmes

Vue générale

Un module du SGBD, **l'optimiseur**, est chargé de :

1. Prendre en entrée une requête, et la mettre sous forme d'opérations
2. Se fixer comme objectif l'optimisation d'un certain paramètre (en général le temps d'exécution)
3. construire un programme s'appuyant sur les index existant, et les opérations disponibles
4. Choisir le bon plan
   - Construire des plans possibles
   - Evaluer (grossièrement) leurs coûts
   - Choisir le meilleurs
   - **Problème**: il y a trop de plans possibles
   - Utiliser des heuristiques pour ne pas explorer tout l'espace des possibilités.

L'optimisation sur un exemple

Considérons le schéma :

- $CINEMA(Cinéma, Adresse, Gérant)$
- $SALLE(Cinéma, NoSalle, Capacité)$

avec les hypothèses :

1. Il y a 300 n-uplets dans CINEMA, occupant 30 pages.
2. Il y a 1200 n-uplets dans SALLE, occupant 120 pages.

Expression d'une requête

On considère la requête : *Adresse des cinémas ayant des salles de plus de 150 places*
En SQL, cette requête s'exprime de la manière suivante :

```
SELECT Adresse
FROM   CINEMA, SALLE
WHERE  capacité > 150
AND    CINEMA.cinéma = Salle.cinéma
```

En algèbre relationnelle

Traduit en algèbre, on a plusieurs possibilités. En voici deux :

1. $\pi_{Cinema}(\sigma_{Capacite>150}(CINEMA \bowtie SALLE))$
2. $\pi_{Cinema}(CINEMA \bowtie \sigma_{Capacite>150}(SALLE))$

Soit une jointure suivie d'une sélection, ou l'inverse.

NB : on peut les représenter comme des arbres.

Évaluation des coûts

On suppose qu'il n'y a que 5 % de salles de plus de 150 places.

1. Jointure : on lit 3 600 pages (120x30); Sélection : on obtient 5 % de 120 pages, soit 6 pages.
   Nombre d'E/S : 3 600 + 120x2 + 6 = 3 846.
2. Sélection : on lit 120 pages et on obtient 6 pages. Jointure : on lit 180 pages (6x30) et on obtient 6 pages.
   Nombre d'E/S : 120 + 6 + 180 + 6 = 312.

$\Rightarrow$ la deuxième stratégie est de loin la meilleure !

Un truc simple qui marche : combiner des opérateurs

Des cascades de projections/sélections peuvent se réécrire en une opération de filtrage (qui ne demande qu’un scan des données)

Un autre truc qui marche: pousser projections/sélections

Pourquoi ? **Ce sont des réducteurs souvent radicaux**

- Relation R et critère de sélection de probabilité p

$$O(\sigma_c(R)) = p * O(R)$$

- n tuples de longueur k et projection sur X de longueur k' $<$ k
- $O(\pi_X(R)) = O(R) * (K'/K)$
  même moins après élimination de dupliquets

Pour les jointures, c'est le contraire

- Relations R_1, R_2 probabilité de jointure p'

$$O(R_1 \bowtie R_2) = p' * O(R_1) * O(R_2)$$

- Dans le pire des cas $O(R_1) * O(R_2)$

Attention : pas toujours vrai (e.g., si p' très petit)

Un autre truc qui marche: réordonner les jointures

Exemple : quels inspecteurs a eu l'aligoté ?

$$Q_1' = \pi_{inom}(\sigma_{nom=aligote}(vin) \bowtie (inspecteur \bowtie test))$$

- Hypothèse: 40 inspecteurs + 500 tests
- $40 \bowtie 500 \longrightarrow 500$ puis $1 \bowtie 500 \longrightarrow 10$

il vaut mieux faire

$$Q_2' = \pi_{inom}((\sigma_{nom=aligote}(vin) \bowtie test) \bowtie inspecteur)$$

$1 \bowtie 500 \longrightarrow 10$ puis $10 \bowtie 40 \longrightarrow 10$

Choisir le bon algorithme

Toutes les opérations peuvent être coûteuse

Il faut choisir le bon algorithme pour la réaliser

Exemple : Trouver l'enregistrement de Marie Martin
Utiliser un index sur nom s'il y en a un

Fin de ce cours : les algorithmes pour le tri et la jointure

Le schéma de la base

- Film (**idFilm**, titre, année, genre, résumé, *idMES*, *codePays*)
- Artiste (**idArtiste**, nom, prénom, annéeNaissance)
- Role (***idActeur, idFilm***, nomRôle)
- Internaute (**email**, nom, prénom, région)
- Notation (***email, idFilm***, note)
- Pays (**code**, nom, langue)

Itérateurs

Tous les systèmes s'appuient sur un ensemble d'opérateurs physiques, ou *itérateurs*. Tous fournissent des interfaces semblables :

1. open : initialise les tâches de l'opérateur ; positionne le curseur au début du résultat à fournir ;
2. next : ramène l'enregistrement courant se place sur l'enregistrement suivant ;
3. close : libère les ressources ;

On appele *itérateurs* ces opérateurs. Ils sont à la base des plans d'exécution.

Plan d'exécution

Un plan d'exécution est un arbre d'itérateurs.

1. Chaque itérateur consomme une ou deux sources, qui peuvent être soit d'autres itérateurs, soit un fichier d'index ou de données ;
2. Un itérateur produit un flux de données *à la demande*, par appels répétés de sa fonction *next*.
3. Un itérateur peut appeler les opérations *open*, *next* et *close* sur ses itérateurs-sources.

La production *à la demande* évite d'avoir à stocker des résultats intermédiaires.

Parcours séquentiel

1. On lit, **bloc par bloc**, le fichier
2. Quand un bloc est en mémoire, on traite les enregistrements qu’il contient.

Sous forme d’itérateur :

1. le *open* place le curseur au début du fichier et lit le premier bloc ;
2. le *next* renvoie l’enregistrement courant, et avance d’un cran ; on lit un nouveau bloc si nécessaire ;
3. le *close* libère les ressources.

Traversée d'index et accès direct

Index : l'itérateur prend en entrée une valeur, ou un intervalle de valeurs.

1. on descend jusqu'à la feuille (*open*) ;
2. on ramène l'adresse courante sur appel de *next*, on se décale d'un enregistrement dans la feuille courante (éventuellement il faut lire le bloc-feuille suivant) ;

Accès direct : s'appuie sur un itérateur qui fournit des adresses d'enregistrement (décrivez les *open*, *next* et *close*).

Calcul du coût par l'optimiseur

Le fichier fait 500 Mo, une lecture de bloc prend 0,01 s (10 millisecondes).

1. Un parcours séquentiel lira tout le fichier (ou la moitié pour une recherche par clé). Donc ça prendra 5 secondes.
2. Une recherche par index implique 2 ou 3 accès pour parcourir l'index, et un seul accès pour lire l'enregistrement : soit $4 \times 0.01 = 0.04$ s, (4 millisecondes).

En gros, c'est mille fois plus cher.

Exemple de plan d'exécution

Pour la requête :
$\pi_{Cinema}(CINEMA \bowtie \sigma_{Capacite>150}(SALLE))$

1. Un itérateur de *parcours séquentiel* ;
2. Un itérateur de traversée d'index ;
3. Un itérateur de jointure avec index ;
4. Un itérateur d'accès direct par adresse ;
5. Un itérateur de projection.

Le plan est exécuté simplement par appels *open; {next}; close* sur la racine (itérateur de projection).

Rôle des itérateurs

Principes essentiels :

1. **Production à la demande** : le serveur n'envoie un enregistrement au client que quand ce dernier le demande ;
2. **Pipelinage** : on essaie d'éviter le stockage en mémoire de résultats intermédiaires : le résultat est calculé au fur et à mesure.

Conséquences : **temps de réponse minimisé** (pour obtenir le premier enregistrement) mais **attention aux plans bloquants** (ex. plans avec un tri).

Tri externe

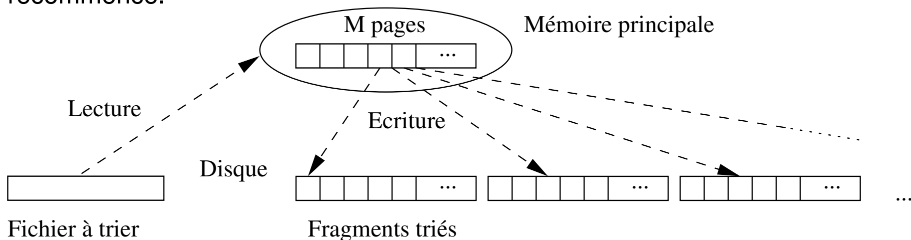
Le *tri externe* est utilisé,

- pour les algorithmes de jointure (*sort/merge*)
- l'élimination des doublons (clause *DISTINCT*)
- pour les opérations de regroupement (*GROUP BY*)
- ... et bien sûr pour les *ORDER BY*

C'est une opération qui peut être très coûteuse sur de grands jeux de données.

Première phase : le tri

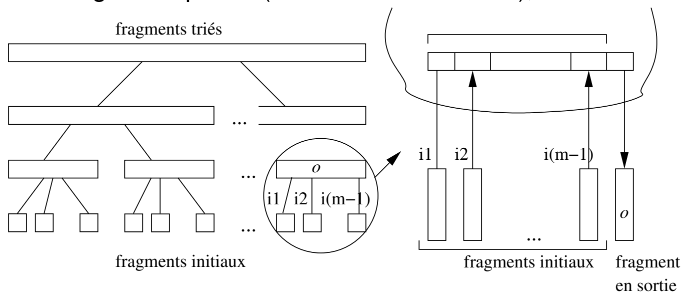
On remplit la mémoire, on trie, on vide dans des **fragments**, et on recommence.



Coût : une lecture + une écriture du fichier.

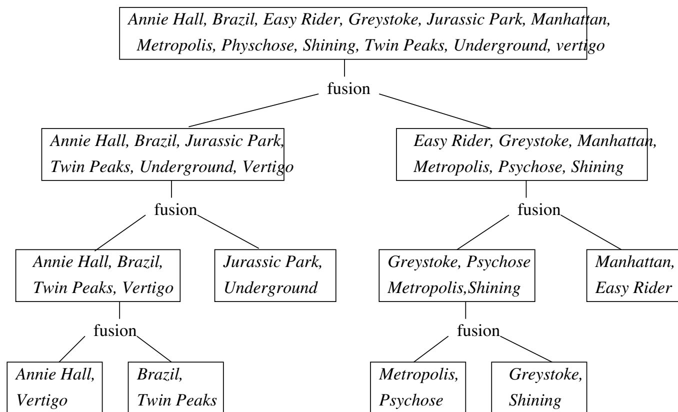
Deuxième phase : la fusion

On groupe les fragments par M (taille de la zone de tri), et on fusionne.



Coût : autant de lectures/écritures du fichier que de niveaux de fusion.

Illustration avec $M = 3$



Essentiel : la taille de la zone de tri

Un fichier de 75 000 pages de 4 Ko, soit 307 Mo.

1. $M > 307Mo$: une lecture, soit 307
2. $M = 2Mo$, soit 500 pages.
   1. le tri donne $\lceil \frac{307}{2} \rceil = 154$ fragments.
   2. On fait la fusion avec 154 pages

   Coût total de $614 + 307 = 921$ Mo.

NB : il faut allouer beaucoup de mémoire pour passer de 1 à 0 niveau de tri.

Avec très peu de mémoire

$M = 1Mo$, soit 250 pages.

1. on obtient 307 fragments.
2. On fusionne les 249 premiers fragments, puis les 58 restant. On obtient F_1 et F_2.
3. On fusionne F_1 et F_2.

Coût total : $1\,228 + 307 = 1\,535$ Mo.

Résultat : grosse dégradation entre 2 Mo et 1 Mo (calcul approximatif).

Principaux algorithmes

Jointure sans index

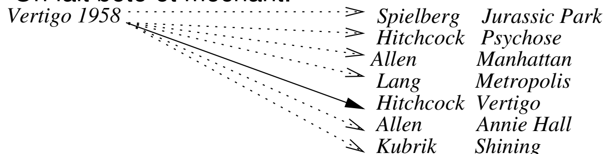
1. Le plus simple : *jointure par boucles imbriquées*
2. Le plus courant : *jointure par tri-fusion*
3. Parfois le meilleur : *jointure par hachage*

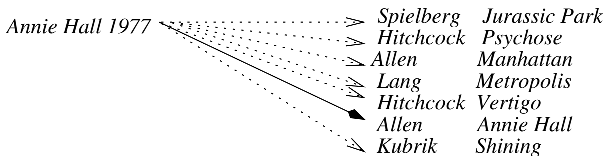
Jointure avec index

1. Avec un index : *jointure par boucles indexée*.
2. Avec deux index : on fait comme si on avait un seul index

Jointures par boucles imbriquées

Pas d'index ? On fait bête et méchant.





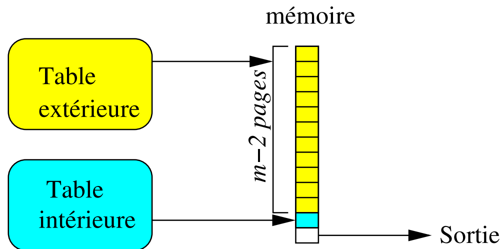
Brazil 1984 ....

Comparaison

Association

Essentiel : la mémoire

On alloue le maximum à la table intérieure de la boucle imbriquée.



Si la table intérieure tient en mémoire : une seule lecture des deux tables suffit.

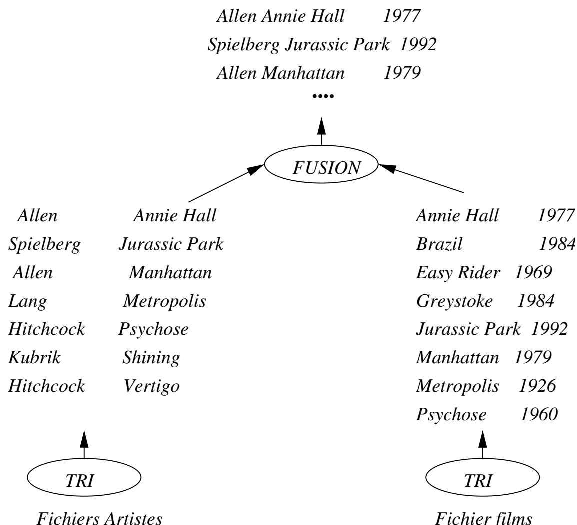
Jointure par tri fusion

Plus efficace que les boucles imbriquées pour de grosses tables.

- On trie les deux tables sur les colonnes de jointures
- On effectue la fusion

C'est le tri qui coûte cher.
Important : on ne peut *rien* obtenir tant que le tri n'est pas fini.

Tri-fusion : illustration



Jointure par hachage

En théorie, souvent le plus efficace.

- Très rapide quand une des deux tables est petite (1, 2, 3 fois la taille de la mémoire).
- Pas très robuste (efficacité dépend de plusieurs facteurs).

Algorithme en option dans ORACLE. **Il est indispensable d'avoir des statistiques**.

Principes de la jointure par hachage

Un peu compliqué... Le principal :

- On hache la plus petite des deux tables en *n* fragments.
- On hache la seconde table, avec la même fonction, en *n* autres fragments.
- On réunit les fragments par paire, et on fait la jointure.

Essentiel : pour chaque paire, au moins un fragment doit tenir en mémoire.

Illustration : phase de hachage

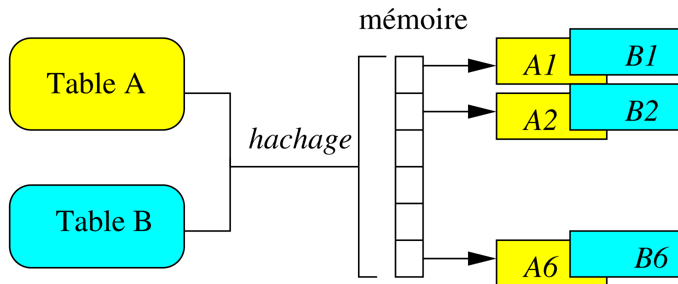
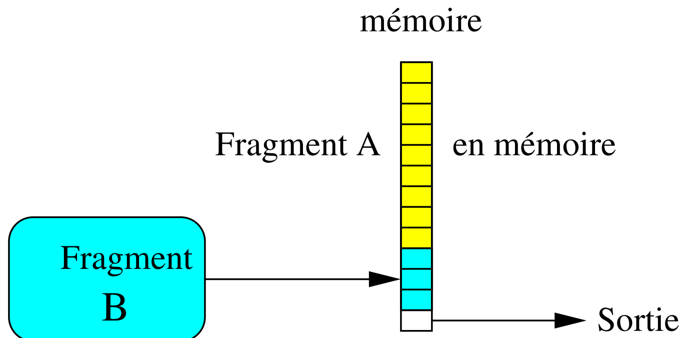


Illustration : phase de jointure



Jointure avec index

Avec un index, on utilise les *boucles imbriquées indexées*.

- On balaye la table non indexée
- Pour chaque ligne, on utilise l'attribut de jointure pour traverser l'index sur l'autre table.

Avantages :

- Très efficace (un parcours, plus des recherches par adresse)
- Favorise le temps de réponse **et** le temps d'exécution

Et avec deux index ?

On pourrait penser à la solution suivante :

- Fusionner les deux index : on obtient des paires d'adresse.
- Pour chaque paire, aller chercher la ligne *A*, la ligne *B*.

Problématique car beaucoup d'accès aléatoires (cf. « Quand utiliser un index »). En pratique :

- On se ramène à la jointure avec un index
- On prend la petite table comme table extérieure.

L'essentiel de ce qu'il faut savoir

Qu'est-ce qu'un plan d'exécution ?

- C'est un **programme** combinant des opérateurs physiques (chemins d'accès et traitements de données).
- Il a la forme d'un **arbre** : chaque nœud est un opérateur qui
  - prend des données en entrée
  - applique un traitement
  - produit les données traitées en sortie

L'essentiel de ce qu'il faut savoir (suite)

La phase d'optimisation proprement dite :

- Pour **une** requête, le système a le choix entre **plusieurs** plans d'exécution.
- Ils diffèrent par l'ordre des opérations, les algorithmes, les chemins d'accès.
- Pour chaque plan on peut estimer :
  - le coût de chaque opération
  - la taille du résultat

Objectif : diminuer le plus vite possible la taille des données manipulées.

Laissons le choix au système !

Bon à savoir : il y a autant de plans d'exécution que de « blocs » dans une requête.

Exemple : cherchons tous les films avec James Stewart, parus en 1958.

```
SELECT titre
FROM Film f, Role r, Artiste a
WHERE a.nom = 'Stewart' AND a.prenom='James'
AND f.idFilm = r.idFilm
AND r.idActeur = a.idArtiste
AND f.annee = 1958
```

Pas d'imbrication : un bloc, OK !

Seconde requête (2 blocs)

La même, mais avec un niveau d'imbrication.

```
SELECT titre
FROM Film f, Role r
WHERE f.idFilm = r.idFilm
AND f.annee = 1958
AND r.idActeur IN (SELECT idArtiste
                   FROM Artiste
                   WHERE nom='Stewart'
                   AND prenom='James')
```

Une imbrication sans nécessité : moins bon !

Troisième requête (2 blocs)

La même, mais avec *EXISTS* au lieu de *IN*.

```
SELECT titre
FROM Film f, Role r
WHERE f.idFilm = r.idFilm
AND f.annee = 1958
AND EXISTS (SELECT 'x'
                    FROM Artiste a
                    WHERE nom='Stewart'
                    AND prenom='James'
                    AND r.idActeur = a.idArtiste)
```

Quatrième requête (3 blocs)

La même, mais avec deux imbrications :

```
SELECT titre FROM Film
WHERE annee = 1958
AND idFilm IN
        (SELECT idFilm FROM Role
         WHERE idActeur IN (SELECT idArtiste
                            FROM Artiste
                            WHERE nom='Stewart'
                            AND prenom='James'))
```

Très mauvais : on force le plan d'exécution, et il est très inefficace.

Pourquoi c'est mauvais

- On parcourt tous les films parus en 1958
- Pour chaque film : on cherche les rôles du film, **mais pas d'index disponible**
- Ensuite, pour chaque rôle on regarde si c'est James Stewart

Ca va coûter cher !!

Exemples de plans d'exécution

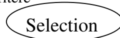
Gardons la même requête. Voici les opérations disponibles :

CHEMINS D'ACCES *OPERATIONS PHYSIQUES*

Séquentiel

Parcours séquentiel

Critère



Sélection selon un critère

Critère

Filtre d'un ensemble en fonction d'un autre

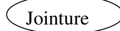
Adresse

Accès par adresse

Attribut(s)

Tri sur un attribut

Critère



Jointure selon un critère

Attribut(s)

Parcours d'index

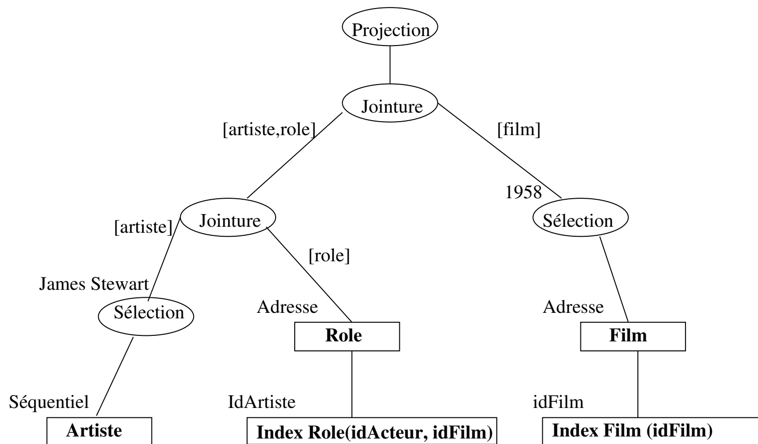
Critère

Fusion de deux ensembles triés

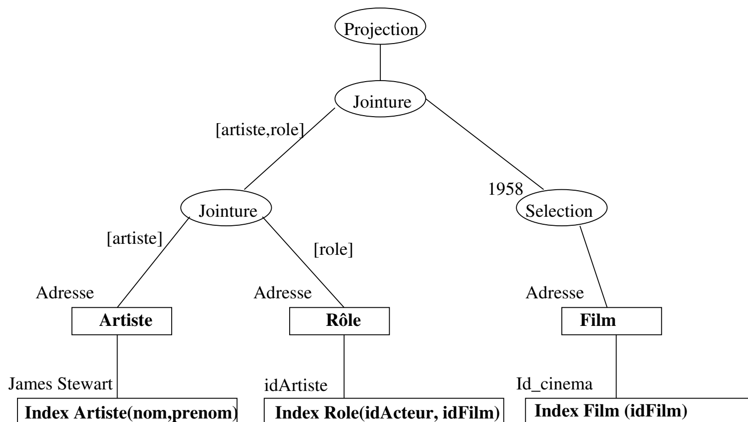
Attribut(s)

Projection sur des attributs

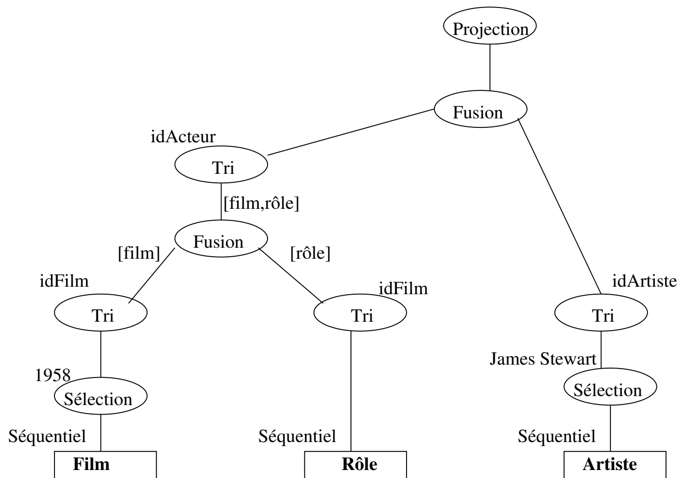
Sans index sur le nom



Avec index sur le nom



Sans index



Faut-il toujours utiliser l'index ?

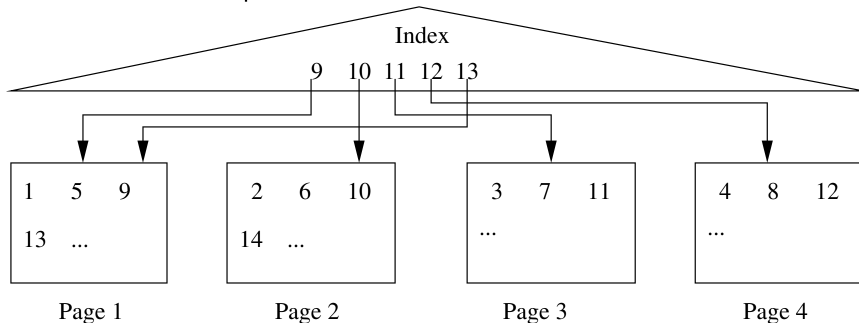
Pour les recherches par clé : oui. Sinon se poser les questions suivantes :

1. Le critère de recherche porte-t-il sur un ou sur plusieurs attributs ? S'il y a plusieurs attributs, les critères sont-ils combinés par des **and** ou des **or** ?
2. Quelle est la sélectivité (pourcentage des lignes concernées) de la recherche ?

Mauvaise sélectivité = contre-performant d'utiliser l'index.

Ce qui peut poser problème

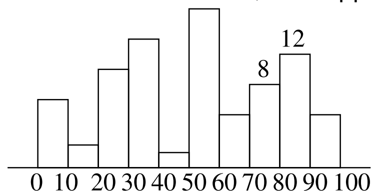
Toutes les recherches par intervalle.



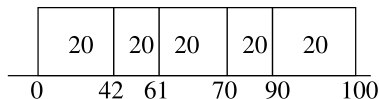
Recherche entre 9 et 13 : mauvais.

Histogrammes

Pour connaître la sélectivité, on s'appuie sur des histogrammes.



(a) Histogramme en hauteur



(b) Histogramme en largeur

Les histogrammes donnent la distribution des valeurs dans une colonne.

Exercice

vin (v#, nom, cépage)
inspecteur (i#, inom)
test (v#, i#, date, note)
Exemple: quels vins ont eu un test négatif ?

$$Q_1 = \pi_{nom}\sigma_{note=F}(vin \bowtie test)$$

$$Q_2 = \pi_{nom}(vin \bowtie \sigma_{note=F}(test))$$

50 vins $\approx$10 tests par vin seulement 3 tests négatifs
Pourquoi préférer Q_2 ?

Exercice

Exemple : quels inspecteurs ont fait échouer l'aligoté ?

$$Q_1 = \pi_{inom}(\sigma_{nom=aligote,note=F}(vin \bowtie (inspecteur \bowtie test)))$$

même technique : **pousser les sélections pour limiter la taille des jointures**

Pourquoi préférer ?

$$Q_2 = \pi_{inom}(\sigma_{nom=aligote}(vin) \bowtie (inspecteur \bowtie \sigma_{note=fail}(test)))$$

Merci